Performance and Power Effectiveness in Embedded Processors—Customizable Partitioned Caches

Peter Petrov and Alex Orailoglu, *Member, IEEE*

***Abstract*—This paper explores an application-specific customization technique for the data cache, one of the foremost area/power consuming and performance determining microarchitectural features of modern embedded processors. The automated methodology for customizing the processor microarchitecture that we propose results in increased performance, reduced power consumption and improved determinism of critical system parts while the fixed design ensures processor standardization. The resulting improvements help to enlarge the significant role of embedded processors in modern hardware–software codesign techniques by leading to increased processor utilization and reduced hardware cost. A novel methodology for static analysis and a microarchitecturally field-reprogrammable implementation of a customizable cache controller that implements a partitioned cache structure is proposed. Partitioning the load/store instructions eliminates cache interference; hence, precise knowledge about the hit/miss behavior of the references within each partition becomes available, resulting in significant reduction in tag reads and comparisons. Moreover, eliminating cache interference naturally leads to a significant reduction in the miss rate. The paper presents an algorithm for defining cache partitions, hardware support for customizable cache partitions, and a set of experimental results. The experimental results indicate significant improvements in both power consumption and miss rate.**

***Index Terms*—Application-specific processors, cache memories, embedded processors, low-power processors, partitioned caches.**

I. Introduction

PROCESSOR performance, power consumption, and deterministic execution time impose significant constraints on modern hardware–software codesigned systems [1]. The partitioning of hardware and software is a major issue in these systems [2]. Implementing larger parts of the system as software results in reduced cost, improved time to market, system maintainability, and flexibility. However, reduced performance and increased power consumption are typically observed due to the nature of the general-purpose processors. Yet processor cores with all their attendant benefits of flexibility, maintainability, and high volumes are natural candidates for further utilization, if the performance and power limitations were to be alleviated. In this paper, we propose a methodology for application-specific customization of the data cache, one of the foremost performance and power stipulating components of modern high-end embedded processors. We present an architecture capable of incorporating the application-specific information in a postmanufacturing fashion, utilizing a reprogrammable data path.

Application-specific customization of embedded-processor microarchitectures is a novel technique that transfers application information to the processor microarchitecture. In this case, the microarchitecture can perform informed decisions as to how to handle various architecture-specific actions. Fundamentally, this approach extends the communication link between compiler and processor architecture by transferring application information directly to the microarchitecture without modifying the existent instruction set. Consequently, traditional mainstream compiler algorithms remain unaffected by the proposed customization technique. The static analysis information transfer is accomplished by utilizing a reprogrammable data path. The reprogrammable implementation we propose allows application changes to be effected in field by loading the new application information in a manner similar to program reloading. The reprogrammability is achieved on a microarchitectural level rather than through gate-level approaches such as field-programmable gate arrays (FPGAs), thus achieving cost-efficient performance and power consumption.

In this paper, we present a methodology for application-specific customization of the data cache of embedded processors for improved power and performance. In a typical embedded environment, the application workload is fixed for extended periods of time, making specialization techniques even more viable. Consequently, more aggressive application-specific optimization techniques can be applied in order to significantly reduce power consumption and increase performance.

Power consumption is becoming an ever more important characteristic in modern embedded applications. The data-cache subsystem constitutes a significant fraction of the total number of transistors in a modern microprocessor and consumes a large part of the total power. An illustrative example is the StrongArm-110 architecture [3], wherein the cache consumes 43% of the total power. It is evident that new techniques are needed to leverage the available application information into a power and performance efficient embedded-processor architecture.

The domain of numerical algorithms, the cornerstone of image and voice processing and various wireless applications, suffers from significantly elevated cache miss rates, as associated algorithms operate on a number of arrays of data with large volume. Consequently, high interference amongst memory references, frequently residing in nested loops, and cache pollution caused by sequential array references are typically observed. Furthermore, a high amount of redundancy exists

Manuscript received April 16, 2001; revised June 29, 2001. This work was supported by an IBM Graduate Fellowship and by the National Science Foundation under Grant 0082325. This paper was recommended by Guest Editor P. Marwedel.

The authors are with the Computer Science and Engineering Department, University of California at San Diego, La Jolla, CA 92093-0114 USA (e-mail: ppetrov@cs.ucsd.edu; alex@cs.ucsd.edu).

Publisher Item Identifier S 0278-0070(01)09990-0.

in reading and comparing tags for references with hit/miss behavior that can be predetermined well in advance because of their regularity. General-purpose cache organizations handle neither of these problems. It is difficult to find an effective solution unless significant application information can be utilized by the cache subsystem. New more sophisticated caching schemes are needed, capable of utilizing application information for the particularities of the specific memory reference pattern.

In this paper, we demonstrate an algorithm that partitions the memory access instructions of a nested loop into groups. Each group corresponds to a set of load/store instructions exhibiting data reuse amongst them and is mapped to a *cache partition* within a *partitioned cache* structure. The size of the cache partitions is determined according to the type of reuse and space needed to exploit it. Moreover, it is shown how the tag array and tag comparisons can be completely turned off for most of the load/store instructions belonging to such partitions as the hit/miss outcome for each instruction in the partition can be determined well in advance regardless of its address tag field. This leads to a significant amount of power reduction and at the same time to a highly improved miss rate. Furthermore, by eliminating cache interference and pollution, a significant reduction in miss rate is achieved, which leads to a reduction in the number of accesses to the power-hungry second level cache or system memory, thus, reducing both power and improving performance.

The proposed technique is highly beneficial on loop nests containing references to several arrays with large amounts of temporal data reuse, but nonetheless suffering from significant data-cache interference and pollution. This is a typical situation for a large number of loops contained within applications for numerical computing and modeling. The benchmarks utilized in our experimental results are representative applications and kernels from the aforementioned application domains.

While we target a single program execution model in our exposition, the proposed methodology can be readily extended to multiprogram environments by mapping the task partitions into separate cache regions. Not only are the benefits from cache partitioning for a single task preserved, but also further possibilities for extending the scope of the proposed framework to controlling the interprogram cache pollution and interferences can be achieved. A separate region of the cache can be dedicated to each task and the task partitions can be inserted within the task cache region. The task cache regions can be organized in the same way as the normal cache partitions and their size determined by the cache utilization of each task. While extensions of the ideas we propose for multiprogramming environments can, thus, be entertained, we concentrate in this paper on illustrating the fundamental ideas of cache partitioning within the essential domain of uniprogramming environments.

We complement this paper with a discussion of an efficient hardware implementation for partitioned caches. Not only is the proposed implementation efficient, but it is also reprogrammable, thus, providing high flexibility to recustomize the embedded processor infield. Hence, the proposed implementation constitutes a unified microarchitectural solution that is not confined to a particular application, but is capable of handling diverse workloads through infield recustomization.

II. Related Work

Various techniques have been proposed in the compiler and computer architecture communities to attack the problem of cache conflicts and cache pollution for data-intensive applications. Loop interchange, skewing, and tiling [4]–[6] all constitute compiler optimization techniques for improving data locality in loop nests. While useful in exploiting a large amount of inherent reuse, controlling the data interference inside the tile is still a significant challenge. Loop transformations introduce extra control code as well, which may preclude their usage in applications with stringent time and power budgets.

Architectural support for distinguishing between memory references that exhibit spatial, temporal, or no reuse whatsoever has been proposed through the introduction of the dual data cache [7]. The approach results in improved cache utilization, but the cache interference problem still remains. A static analysis approach for avoiding data-cache interference is presented in [8]. Therein, memory references that can cause cache conflicts are annotated as noncacheable. Additionally, a cache volume (CV) analysis for facilitating the feasibility of exploiting particular data reuse is described.

Various optimization techniques for reducing energy dissipation in the cache subsystem have been proposed recently. A power optimization technique applied during behavioral synthesis for memory intensive applications has been presented in [9]. The behavior of the memory access patterns is utilized to minimize the number of transitions on the address bus and decoder, thus reducing power consumption. In [10], a small L0 instruction cache is proposed to store the most frequently executed basic blocks in order to reduce the energy dissipation of the larger L1 I-cache. A technique for turning off associativity ways in a set-associative cache architecture is proposed in [11]. The proposed approach represents a tradeoff between power reduction versus performance. A set of memory array implementation techniques for minimizing power in the cache subsystem is presented in [12]. The authors propose subbanking, bitline segmentation, and multiple line buffers in order to minimize the energy dissipated when accessing the memory array. A fast register-transfer level power-estimation methodology has been presented in [13]. The methodology accurately estimates the power consumption for both the controller and data path. A new energy estimation framework for microprocessors has been proposed recently in [14]. The simulation environment employs a transition-based power model and rapidly achieves highly precise power estimations.

III. Motivation

A typical numerical algorithm consists of matrix and/or array operations grouped in several nested loops. For example, Fig. 1 shows an excerpt from the *swim* SPEC95fp benchmark. One can notice that the references to U and V exhibit only spatial locality. All references to the array PSI utilize spatial locality as well, but there is also a temporal locality between $\mathrm{PSI}[i, j+1]$ and $\mathrm{PSI}[i+1, j+1]$. These references reuse a row of the matrix PSI along the outer loop i. The references $\mathrm{PSI}[i, j+1]$ and $\mathrm{PSI}[i+1, j]$ will always result in hits if the data brought by $\mathrm{PSI}[i+1, j+1]$ is judiciously preserved in the cache. Not

```
for i=1 to N
 for j=1 to M
  U[i,j+1]=-(PSI[i+1,j+1]-PSI[i,j+1])/DY;
  V[i+1,j]= (PSI[i+1,j+1]-PSI[i+1,j])/DX;
 end for
end for
```

Fig. 1. Excerpt from the *swim* benchmark.

only do such judicious data preservations promote performance and power improvements through the evident miss rate reduction, but, furthermore, a large number of power-consuming tag operations can be completely avoided, if knowledge regarding whether a particular array reference will indeed hit in the cache can be captured.

Consider, for example, the write references to U and V. They are brought into the cache and a new cache line is used when the previous one is filled. This situation occurs when an array (or matrix) is traversed sequentially. In this type of access, only spatial locality exists and it is not necessary to use more than one cache line to capture this locality. Such reference behavior leads to significant cache pollution and interference with the remaining working set, as using more than one cache line leads to no benefits in terms of reuse. The inherent spatial reuse can be exploited using a single cache line, if a more sophisticated caching technique were to be utilized. A similar problem of interference may occur when temporal reuse with overlapping accesses in the data cache amongst more than one array exists; the overlapping accesses may prevent the temporal reuse from being utilized.

A conventional cache organization suffers from the inability to distinguish different types of reuse along the loop iterations. All references are treated identically; thus, significant interference between unrelated data arrays and cache pollution is introduced. Caching the data intelligently by avoiding interferences and pollution would be highly beneficial in terms of both performance and energy dissipation gains. At the same time, a significant number of tag reads and comparisons can be eliminated if the interference are to be avoided. For example, it is evident that the $\mathrm{PSI}[i+1,j]$ and $\mathrm{PSI}[i,j+1]$ would always hit in the cache because of $\mathrm{PSI}[i+1,j+1]$. Consequently, a significant power reduction can be achieved through elimination of the tag operations and reduction of the miss rate (which will lead to fewer accesses to the even more power-consuming L2 cache or main memory). Solving these problems of performance and power simultaneously is a rather difficult task for a general-purpose cache controller.

The sole purpose of the tag in the cache subsystem is to validate whether the data being accessed resides in the cache, yet the tag array operations are some of the most power-consuming activities within the cache. Thus, exploiting regularities and eliminating a large number of tag reads and comparisons would provide significant benefits in terms of reducing power consumption. For a large number of data-intensive and numerical loops, the miss/hit behavior of the memory references can be analyzed well in advance. As the example in Fig. 1 suggests, intelligent avoidance of the cache interference on the basis of advance application knowledge obviates most of the tag operations.

The group of references to the matrix PSI exhibits temporal reuse along iteration i. Namely, the usage of rows $\mathrm{PSI}[i]$ and $\mathrm{PSI}[i+1]$ is overlapped in the computation process. In order to exploit this reuse, the rows need to be protected from interferences by U and V, which might affect them during iteration j. Having isolated the references to PSI from other unrelated references, the tag operations associated with the references to $\mathrm{PSI}[i]$ can be completely eliminated. Due to interference and volume limitations, a conventional cache organization fails to exploit the reuse and to eliminate the unnecessary tag operations.

The fundamental difficulty in achieving higher reuse utilization and power efficient usage of the tag arrays is the inability of a conventional cache to prevent interferences amongst unrelated references, thus avoiding cache pollution. Utilizing application-specific information can result in separation of interfering references and elimination of the cache pollution introduced by references with no temporal, but only spatial reuse. References requiring a larger number of cache lines for temporal locality exploitation can benefit from the partitioned cache. At the same time for the references, thus, separated, it can be observed that their hit/miss cache outcomes can be analyzed in advance, thus, obviating both tag lookups and comparisons.

Cache interference and pollution problems can be resolved in an application-specific environment, wherein more precise information about the inherent reusability can be provided to the cache controller. If the memory instructions were to be grouped according to the inherent reuse characteristics amongst them and each group subsequently mapped to a dedicated cache partition, behaving in the same way as a distinct cache, all conflicts would be obliterated and redundant tag manipulations completely avoided. The size of each cache partition can, thus, be reduced to no larger than the minimal sufficient size for exploitation of the inherent reuse for that particular group of instructions.

IV. Partitioning Analysis

The proposed partitioning analysis utilizes information about the type of reuse of each reference. A formal methodology for determining the reuse type of array references with affine indexes is presented in [5]. Since the methodology we propose utilizes information about reuse type in a loop nest, we briefly review the relevant terminology.

A memory reference instruction is said to have *self-temporal* (*st*) reuse if in a subsequent loop iteration it accesses the same memory address. A *self-spatial* (*ss*) reuse refers to an instruction that accesses data inside a single cache line in two subsequent loop iterations. Two load/store instructions are said to have *group-temporal* (*gt*) reuse if both of them access the same memory address; they are denoted as *group-spatial* (*gs*) if both access memory addresses that map to the same cache line.

The reuse type varies across loop dimensions. A reuse occurs in a particular loop dimension and is qualified by the number of iterations within which the reuse is exploited.

Under the assumption that the reference analysis operates on perfectly nested loops where all the references to a data array reside in the innermost loop dimension and have indexes with the same multiplicative coefficient,[1] it is straightforward to guarantee the absence of any cache consistency problems. Further affine-linear analysis can be performed to identify whether a consistency problem can occur in the cases of imperfectly nested loops and for nonidentical multiplicative coefficients. If the absence of consistency problems is, thus, ascertained, the same partitioning methodology mentioned above can be applied. In the case of possible consistency issues, the corresponding references are not further considered for partitioning. In any case, references with nonidentical multiplicative coefficients in indexing the same array are quite uncommon in practice and in none of the benchmarks that we have considered in our experimental study were any such instances encountered.

By isolating a group of load/store instructions from other unrelated and possibly interfering groups of memory references, the detrimental effects of cache conflicts can be minimized. The grouped instructions can be considered as a set composed of a *leading* reference and several *trailing* references. The *leading* reference does fetch data from memory, but misses only once per cache line. All the *trailing* references invariably hit in the cache. Consequently, no tag operations are needed for the trailing references within a partition. Furthermore, if the data access is single-strided, the leading reference would miss only in the beginning of a cache line. Therefore, in this case, no tag operations are needed for the leading reference as well.

A. Algorithm Fundamentals

We capture the information about the inherent reuse for a particular loop dimension by constructing a data reuse graph (DRG). Each node in the DRG corresponds to a particular load/store instruction or to an already formed group. The edges in the DRG represent data reuse between the corresponding nodes. Each edge is annotated with the particular type of reuse it represents. Additionally, an integer k is associated to every temporal reuse denotation, representing the number of iterations needed to exploit the temporal reuse denoted by the edge. The number of iterations in turn determines the CV needed to exploit the reuse.

The optimal cache partition size, CV, varies depending on the reuse type. It is evident that a spatial reuse necessitates only a single cache line. In the case of temporal reuse though, a fixed (but varying amongst memory instructions) number of cache lines are needed in order to exploit the reuse and prevent interference.

For example, a pair of loads $A[i]$ and $A[i+k]$, representing an array traversal with loop index i, requires $\lceil k/l \rceil + 1$ cache lines on a cache with line size l to exploit the group-temporal reuse between these references. If the reuse occurs in the outer loop iterations, such as $\mathrm{PSI}[i+1, j+1]$ and $\mathrm{PSI}[i, j+1]$ from the example in Fig. 1, all data referred along iteration j by both instructions need to be preserved. Intuitively, this corresponds to keeping the $\mathrm{PSI}[i]$ row in the cache while $\mathrm{PSI}[i+1, j+1]$ is "prefetching" the next row.

[1]Such as, for example, the reference pair $A[i]$ and $A[i+3]$, but not $A[i+2]$ and $A[3*i+2]$.

```
for i=1 to N
  f(A[i],A[i+1],A[i+4],A[i+7],A[i+8],A[i+12]);
  g(B[i],B[i+2]);
  h(C[i],C[i+3]);
end for
```

Fig. 2. Example of group temporal reuse.

B. Algorithm Overview

The objective of the partitioning algorithm is to group memory instructions with data reuse amongst them and map this group to a cache partition with appropriate size. From a DRG perspective, this implies selecting DRG edges and grouping the neighboring selected edges together into partitions. While each edge selected provides a constant benefit in capturing a reuse, the cost in terms of CV to accommodate an edge varies. Consequently, the objective of the algorithm is to maximize the number of selected DRG edges.

The algorithm starts from the innermost loop dimension, as the frequency of the data reuse there is maximal. The data reuse frequency wanes as the traversal proceeds toward the outermost dimension. Therefore, the algorithm proceeds iteratively on the loop dimensions starting from the innermost loop.

An example of group-temporal reuse can be seen in Fig. 2. We assume a cache line size of one word to simplify the explanation. Fig. 3(a) shows the DRG for the example in Fig. 2. The number of cache lines needed to capture the corresponding group reuse is therein shown. The initially appealing solution of a direct greedy approach is unfortunately inadequate in determining the optimal number of partitions, as illustrated immediately hereafter. If we assume an available CV of nine cache lines, a direct greedy approach leads to the result shown in Fig. 3(b). It is evident though that a solution consisting of a single partition covering all references to the array A, but the last one of $A[i+12]$ is superior in that it covers instead four temporal reuses in A and furthermore utilizes all nine cache lines exactly. The latter solution evidently constitutes an improvement, as a combination of two edges with a common node in a single partition "saves" the storage of the overlapping node. As the counterexample illustrates that the straightforward greedy approach is inadequate in attaining a consistently optimal result, we proceed to outline an improved model of representation that takes into account the cause of the inadequacy, to wit, the overlap between the nodes in the DRG.

C. Overlap Considerations

As discussed in Section IV-B, the inherent overlap of CV for two edges in the DRG with common nodes prevents the direct greedy approach from consistently finding the optimal solution. We present an efficient algorithm that quickly finds an optimal solution for a loop dimension with unit overlap between the nodes in the DRG. The partitioning methodology consists of two steps, the first step constituting a mapping to a group representation that precisely denotes the semantics of overlap and the latter step performing an algorithm for the optimal solution of the problem posed.

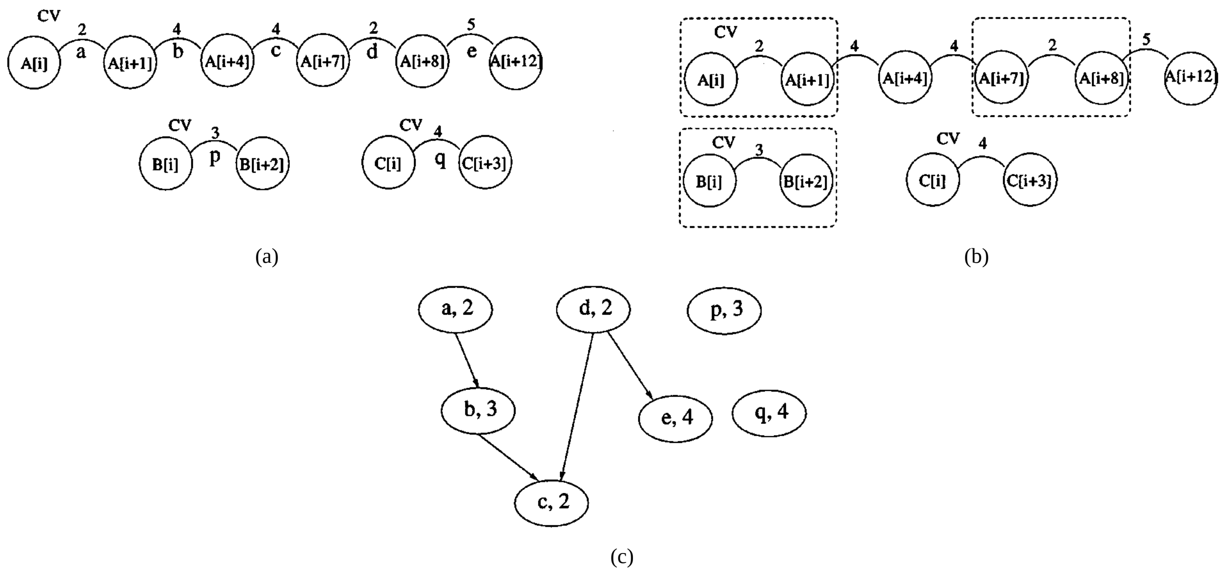


Fig. 3. DRG and ODT for group-temporal reuse. (a) DRG. (b) Greedy nonoptimal DRG partitioning. (c) ODT.

In the first phase, the algorithm separates the connected edges while keeping track of the overlap by building overlap dependence trees (ODTs) as shown in Fig. 3(c). The purpose of the ODT is to represent the overlap dependencies amongst the edges in the DRG and to show the exact CV needed to cover a particular edge in the DRG after removing the overlap. Each node in the ODT corresponds to an edge in the DRG. The edges in the ODT represent the overlap relation between the corresponding edges in the DRG. The following pseudocode describes the ODT construction algorithm formally.

```
repeat
 e = FindEdge with min CV in DRG;
 Decrement CV of Neighbors(e);
 ODT(e) = CreateNode in ODT for e;
 for all n in Neighbors(e) and n in ODT
  Connect ODT(e) as a child of ODT(n);
until no edges left in DRG
```

A tie in the minimal CV value in the DRG can be partially resolved by giving reduced priority to the edges on the boundary of the reuse chain; any remaining ties can be resolved randomly with no impact on optimality. The ODT for the example in Fig. 2 is shown in Fig. 3(c). Each node in the ODT is annotated with the updated CV.

D. Generation of Optimal Solution

Given the ODT, the purpose of the algorithm is to find the maximum subset of nodes subject to the constraints that the total available CV not be exceeded and that overlap dependences be preserved. The following pseudocode defines this part of the algorithm.

```
procedure FormPartitions
CacheVolume = TotalCacheVolume;
while true
 V = {v1, ..., vn} = FindNodes with min CV in
 ODT;
  for all vi in V & vi is a root
   if CV(vi) > CacheVolume then return(Par-
   titions);
   Select vi; ODT -= {vi}; V -= {vi};
   CacheVolume -= CV(vi);
  end for
  repeat
   Find vk in V with minimal
    number of ancestral nodes;
   for w in Path to vk from the root
    if CV(w) > CacheVolume then return(Par-
    titions);
    Select(w); ODT -= {w}; V -= {w};
    CacheVolume -= CV(w);
   end for
  until V is empty
end while
end procedure
```

While minimal CV nodes are invariably root nodes of the graph initially, minimal nodes can be found at arbitrary positions in the ODT in later iterations. This nonstraightforward case, though, can be handled by observing that ancestors of the minimal nonroot node cannot exceed it by more than one unit. Selecting the root from the tree that contains a minimal node with the smallest number of ancestors maintains optimality while paving the route for eventually incorporating the truly minimal CV node into the partition. The algorithm terminates when there remains insufficient CV to accommodate any further reuse.

Informally, the correctness of the algorithm can be illustrated by observing that the more general problem with no edge overlap is optimally solved by a greedy approach. The ODT ensures that the dependence between the overlapped edges in the constructed solution is preserved during the application of the greedy steps. While the prioritization of nodes in the ODG may violate the optimality of the greedy approach, the particular property of the ODG that ancestors of minimal nonroot node cannot exceed it by more than one unit ensures the optimal applicability of the greedy like approach.

The algorithm that we have presented exhibits a linear execution time in terms of the number of edges in the DRG. The input size corresponds to the number of load/store instructions within the loop nest, which in practice is limited to 40–50. Consequently, the running times of the algorithm are exceedingly small, which makes this algorithm particularly useful for design space exploration in hardware–software partitioning schemes.

In the innermost loop iteration, the overlap consists of one cache line, i.e., a unit overlap. When the algorithm proceeds onto the next (outer) loop dimension, the overlap between the nodes in the DRG corresponds to the size of the previous loop dimension, i.e., the memory volume needed for an iteration of the inner loop dimension. In terms of matrix traversal, the overlap corresponds to one row from the matrix. When proceeding to the outer loop dimensions, though, the normalized overlap might be slightly less than a unit because there may be a partition formed in the embedded loop dimension of size negligible compared to the new dimension overlap unit. This deviation can in rare cases result in a globally nonoptimal partition definition. For the innermost loop iteration, of course, which typically contains most of the data reuse, the overlap between the DRG edges is always one cache line as shown in Fig. 3(a); hence, the practical optimality of the algorithm. The experimental studies undertaken have consistently identified solutions not only locally, but also globally optimal, thus, further emphasizing the superiority of the algorithm in an empirical sense.

V. Implementation

The proposed partitioning methodology requires special hardware support from the cache controller. The hardware needs to be able to capture the information provided by the compiler about load/store instruction partitioning and to effectively map these references to the corresponding part of the partitioned cache. Information also needs to be captured regarding the memory references that will not need to perform tag reads and comparisons.

In this paper, we consider direct-mapped cache organizations, but the approach can be generalized to set-associative structures as well by mapping each partition across all the associativity ways. The straightforward implementation generalization relies on the same architecture that we describe in this section. While set-associative organizations can be used within a partition, no strong rationale for such an organization exists, as the traditional benefits of miss rate reduction in set-associative caches do not hold in the proposed partitioned caches due to the lack of conflict misses within a partition.

In either case, the cache is virtually partitioned into subcaches, each of them accommodating a group of load/store instructions. Each cache partition is identified by two parameters: the number of cache lines (size) and offset (position) in the original cache array.

A. Identifying the Cache Partitions

In order to address a particular cache partition as a distinct cache, a slight modification of the traditional cache indexing

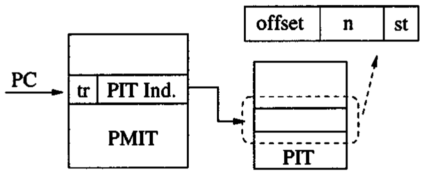


Fig. 4. Partition identification logic.

scheme needs to be effected. Depending on the size of the cache partition, the *cache index* part of the address is divided into two parts. If the size of the corresponding cache partition is 2^n, then the n least significant bits from the *cache index* are used to form the new index. The remaining most significant bits from the cache index are replaced by a constant in the newly formed *cache partition index*. The value of this constant determines the *offset* of the cache partition in the original cache.

The above reasoning evinces that each cache partition with size 2^n is identified by a pair of numbers $\{\text{offset}, \text{cache partition index size}(n)\}$. The *cache partition index* is formed by concatenating the *offset* and the n least significant bits from the cache index.

The hardware support for the partitioned cache has to resolve the following problems: identification of the mapping between a memory instruction and a particular cache partition; identification of the *trailing* load instructions that can avoid the tag read and comparison and calculation of the *cache partition index* using the pair of numbers identifying the partition.

The approach targets loop nests. Therefore, a solution is needed that can efficiently handle the load/store instructions per loop nest. The partition mapping identification is achieved by a hardware architecture utilizing two tables: the partition mapping identification table (PMIT) and the partition identification table (pit). Fig. 4 presents this structure. The PMIT is used to define the mapping between load/store instructions and cache partitions. The PMIT is indexed using the least significant PC bits of the load/store instruction from the loop. The size of the PMIT corresponds to the total number of instructions within the loop nest. In practice, the size of a loop nest in data intensive applications is rarely large, thus, leading to implementations with a small number of entries. When a load/store instruction is decoded, the PMIT is indexed with the least significant bits of the PC. An entry in PMIT contains a value that represents an index into the PIT, which in turn contains a partition-defining information. An additional bit *tr* is also stored in the PMIT entry to indicate whether the instruction is a *trailing* reference for the partition. The main purpose of this organization is to avoid associative lookups, which are expensive in terms of power. The tables described above are directly indexed; their size, negligible compared to the size of the tag memory arrays, ensures that no significant amount of energy dissipation is introduced.[2] The timing impact on the access time is discussed later in this section.

All memory references in a loop nest that are left unpartitioned by the reference analysis are mapped to a dedicated cache partition. This special partition is treated in the same way as the remaining cache partitions. The only difference is that no tag optimizations are performed for it. One of the entries in PIT is

[2]The impact of the tables on power consumption is nonetheless fully accounted for in the experimental studies of Section VI.

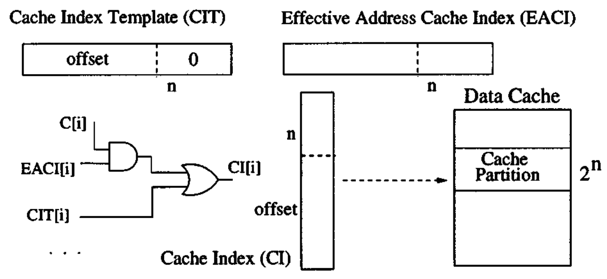


Fig. 5. Partition cache index calculation.

dedicated for this special partition and all unpartitioned memory references are directed to this entry through the PMIT.

The partition information is stored in the PIT. A PIT entry is shown in Fig. 4. The first two fields contain the partition-defining information—*offset* and n as explained earlier. The information from this table is used to calculate the partition cache index. The *st* field specifies whether the access stride for the partition is one. If this is the case, the *leading* reference misses only in the beginning of the cache line; hence, the hit/miss signal can be determined from the least significant bits of the address corresponding to the cache line index.

PMIT and PIT contents are loaded onto the processor at the same time when the code of the embedded application is stored in the main memory. This reprogrammable hardware solution facilitates flexibility across various embedded applications and across versions of the same application.

B. Computing the Cache Index

The lookup into the PMIT and PIT is the first step in determining the *cache partition index* and is performed early in the pipeline, thus, not affecting the cache access time. Right after the load/store is decoded, the lookup is performed in parallel with the effective address calculation. Fig. 5 shows the implementation of the *cache partition index* calculation. The cache index template (CIT) and control signals $C[i]$ are computed before the actual cache access pipeline stage using the partition information found in PIT. The CIT is defined as having the *offset* value in its most significant bits and zeroes in its n least significant bits resulting in control signals $C[i]$ defined as $C[i] = 1$ for $0 \leq i < n$ and $C[i] = 0$ for $i \geq n$. The effective address cache index (EACI) is the traditional cache index field in the effective address. The cache index (CI) is computed using the simple combinatorial logic depicted in Fig. 5. The delay of the two gates shown in this figure is the sole, evidently insignificant, increase in the path delay of the cache access data path.

C. Tag Support for the Partitioned Cache

Since the cache partitions are subsets of the original cache array, the tag field needed to accommodate a particular cache partition will have larger size. For example, if a partition occupies half of the cache array, the tags associated to this partition should have width one bit larger than that of the original tags as the cache block size is unchanged. Yet this larger tag is needed only for the unpartitioned load/store instructions and for the *leading* references for the partitions in the case of nonsingle stride access patterns.[3] Various approaches for implementing these wider tags can be entertained. One possibility is to have an additional extended tag array that can be disabled at the bitlines level. For example, if only two additional tag bits are required for a particular partition (with size one fourth of the cache array) in a loop nest, only two bitlines will be enabled in the extended tag array. Although optimal in minimizing the tag bits usage, this approach requires a somewhat complex control logic. The approach that we have followed in our experimental study, shown in Fig. 6, utilizes again an additional *extended tag array*, but is not controllable on bitline level. The size of the *extended tag array* is determined by the size of the *minimal cache partition* allowed for the particular implementation. Each time an access in a cache partition is performed and this access requires a tag comparison, the whole tag field from the *extended tag array* is used. The width of the extended tag array is four and five for 8- and 16-kB DM caches, respectively, with a minimal partition size of 32 cache lines.

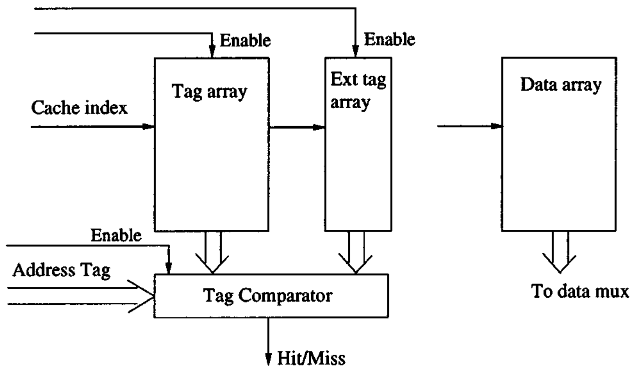


Fig. 6. Extended tag array and enabling logic.

D. Impact on the Cache Access Time

The mapping between the load/store instructions and the cache partitions is identified by indexing into PMIT and PIT. This task can start right after the load/store instruction is decoded. Computation of the template CIT and the control signals $C[i]$ shown in Fig. 5 is independent of the effective address of the memory instructions; hence, the two computations can be overlapped in the pipeline. Only the computation of the final cache index is performed after the effective address computation stage. Consequently, the access to the PMIT and the PIT tables is decoupled from the cache access logic, which happens typically later in the pipeline. In the cache access stage, only the addition of the final cache index computation logic is introduced. To compute the final cache index, the combinatorial logic from Fig. 5 needs to be utilized. Effectively, only the delay of the two gates is added to the path that determines the cache access time. Depending on the critical paths of the pipeline stages for cache access and effective address computation, this small logic can be balanced amongst these two pipe stages.

[3]For the cold cache misses, the invalid bit constitutes an identification for cache miss.

TABLE I
CACHE PERFORMANCE FOR THE BASE CONFIGURATIONS

	8K DM cache		8K 2-SA cache		16K DM cache		16K 2-SA cache		Total
	#Misses	MR	#Misses	MR	#Misses	MR	#Misses	MR	#references
swim	2,542,560	28.51%	2,050,160	22.99%	2,539,666	28.47%	2,044,689	22.92%	8,919,264
tri	295,920	39.20%	237,601	31.48%	295,904	39.20%	237,579	31.47%	754,831
ej	2,833,890	31.84%	1,613,469	18.13%	2,830,276	31.80%	1,606,027	18.05%	8,899,826
sor	8,690	2.22%	8,690	2.22%	8,682	2.22%	8,682	2.22%	391,132

E. Covering Multiple Loop Nests

The proposed methodology works on an individual loop-nest level. A typical numerically intensive embedded application contains several loop nests. In order to be able to perform the cache partitioning methodology for all loop nests, multiple partition mappings associated with each loop nest have to be stored. A straightforward solution consists of having multiple PMITs. Each time the application finishes with one loop nest and proceeds to another, a switch between the PMITs needs to be performed. An alternative approach is to have larger PMIT and PIT tables, multibanked for energy savings; by redefining the index mapping into them, an effective switch to a new partition configuration can be effected.

During a switch between loop nests, the cache content has to be invalidated because a new partition mapping is defined. The switch of partition mappings when proceeding to different loop nests can be controlled by software using a special control register that determines the active partition mapping.

VI. Experimental Results

In our experimental work, we evaluate and analyze the ability of the partitioned cache to reduce both the data-cache miss rate and the amount of energy consumption. We compare the performance and power reduction parameters of the partitioned cache structure against a number of typical L1 cache configurations in modern embedded processors. Specifically, our base configurations include 8- and 16-kB L1 data-cache configurations, both direct-mapped and two-way set associative. All cache configurations contain blocks with a size of four words. Increased cache associativity is a classical approach for reducing the conflicts in the cache; we, therefore, include in our experimental study two-way set-associative cache configurations. Two configurations for a partitioned cache, 8- and 16-kB direct mapped with line size of four words, are examined.

The SimpleScalar toolset [15] has been used to examine the cache behavior for the baseline cache architectures and the partitioned cache has been modeled by a specially developed tool, utilizing traces produced by SimpleScalar.

The Cacti tool [16] has been used to obtain the energy consumption for all the data-cache components. A technology process of 0.35 μm and 2.6 V is assumed. The Cacti tool models in detail all the cache components.[4] We model the PMIT and PIT tables as static random access memory blocks.

[4]In utilizing this tool, we excluded for all partitioned instructions the energy consumed in the tag address decoder, tag wordline selection, tag bitlines precharge, tag sense amps, and the tag comparator in order to reflect the fact that the tag operations for these references are turned off in the architecture we propose.

TABLE II
PARTITION INFORMATION

	Loop1			Loop2		
	MI	PMI	# Part	MI	PMI	# Part
swim	25	19	3	31	20(16)	4(3)
tri	12	6	2	10	6	2
ej	19	11	3	-	-	-
sor	6	6	1	-	-	-

The PMIT contains 64 entries, each of them with a size 4 bits, while the PIT contains eight entries with size 9 and 10 bits for 8-kB and 16-kB data-cache configurations, respectively. The minimal partition size is set to 32; consequently, the width of the extended tag array is 4 and 5 bits, respectively. The energy for the main memory is based on the data presented in [17] and assumes 4.95 nJ per access.

The following benchmarks are used in our experiments: 1) swim benchmark (swim), part of the SPEC95fp benchmark suit, characterized by a high cache miss rate due to a large amount of interference [8]; 2) tri-diagonal system solver (tri), a fundamental part of *tomcatv* SPEC95fp benchmark and a major contributor to the high miss rate for the *tomcatv* benchmark, with matrix size of 128 $\times$ 128; 3) extrapolated Jacobi-iterative method (ej) [18] on a 128 $\times$ 128 grid; 4) successive over-relaxation (sor) [5], [18] on a matrix with size 256 $\times$ 256.

Table I shows the miss rate for the base cache configurations. The high miss rate for *swim, tri,* and *ej* is due to the high amount of interference and cache pollution. The *sor* does not exhibit a high miss rate, as the algorithm works only on a single matrix. Nevertheless, the power reduction in terms of eliminating almost all of the tag operations is significant.

Table II contains information about the partitions for each benchmark. The first column in each subtable shows the total number of memory instructions (MI). The second column gives the number of partitioned memory instructions (PMI), corresponding to the instructions for which no tag operations are performed. The third column displays the total number of partitions for the loop nest, not counting the special partition for the remaining unpartitioned memory instructions. The data pertains to both 8- and 16-kB caches; for the few cases when the results differ for the two caches, the 8 kB results are shown in parentheses.

Table III shows the results for the partitioned cache. Within each cache partition, the number of conflict misses is reduced to zero. This follows directly from the way the cache partitions are formed. Consequently, only the cold misses for the references within the partitions need to be considered in terms of cache miss behavior. Furthermore, one can observe that only the *leading* reference of the partition exhibits cold misses, due

TABLE III
PARTITIONED CACHE MISS-RATE RESULTS

	8K DM p-cache		16K DM p-cache	
	#Misses	MR	#Misses	MR
swim	1,533,664	17.19%	1,257,046	14.09%
tri	173,044	22.92%	165,142	21.88%
ej	1,254,990	14.01%	1,245,372	13.99%
sor	8,690	2.22%	8,682	2.22%

TABLE IV
ENERGY CONSUMPTION (mJ) FOR 8-kB CACHES

	DM	2SA	PC-T	%DM	%2SA	PC	%DM	%2SA
swim	25.75	28.11	22.34	13.24	20.53	20.59	20.04	26.68
tri	2.58	2.70	2.08	19.38	22.97	1.99	22.87	26.30
ej	27.16	25.90	21.49	20.88	17.03	19.19	29.35	25.91
sor	0.62	0.83	0.71	-14.52	14.46	0.54	12.91	34.94

TABLE V
ENERGY CONSUMPTION (mJ) FOR 16-kB CACHES

	DM	2SA	PC-T	%DM	%2SA	PC	%DM	%2SA
swim	31.07	33.02	26.54	14.68	19.63	24.59	20.86	25.53
tri	3.03	3.11	2.61	13.87	16.08	2.40	20.80	22.83
ej	32.46	30.80	23.99	26.10	22.12	24.27	25.24	21.21
sor	0.85	1.05	0.97	-14.11	7.62	0.75	13.77	28.58

to its inherent role in bringing data into the partition for subsequent spatial reuse for itself and temporal reuse for the *trailing* references. We can conclude, therefore, that the conflict misses for the arrays targeted by partitions have been completely eliminated and reduced to zero; only a single memory reference within the partition exhibits cold misses when bringing in a new cache line.[5]

The reduced miss rate has two major implications. First, it leads to performance improvement and second, it results in a significant power reduction from the L2 cache or main memory. The reduction in energy dissipation is proportional to the reduction in the miss rate.

Tables IV and V show the energy consumption results (in millijoules) for 8- and 16-kB caches, respectively. The first two columns represent the energy consumption for direct-mapped and two-way set associative caches. The third column (PC-T) shows the energy for the partitioned cache, but with no tag optimization. The next two columns display the power improvements in percentages, compared to the direct-mapped and the two-way set associative caches. The sixth column (PC) represents the energy consumption for the partitioned cache including both the miss rate reduction and the tag optimizations. Finally, the last two columns present the total improvement in percentages. The energy consumption improvements vary from 14% to 35%. The negative result for *sor* in comparison to a direct-mapped cache in the case of no tag optimizations is due to the lack of miss rate reduction for this benchmark, the extended tag array, and the slight overhead of the partitioned cache in terms of PMIT and PIT. This benchmark is a worst case scenario in terms of power for a partitioned cache with no tag-related power optimization support, since it includes a large number of array references utilizing the extended tag array with no decrease in the miss rate. Nevertheless, when the tag optimizations we propose are included, the energy savings for this benchmark are not only significant, but exhibit the highest level of power reductions for both 8- and 16-kB caches.

[5]This observation is generally true for single-strided loop traversals or a non-single-strided, but within the cache line size traversal, which is the case in all the benchmarks we have utilized in our experimental results. In any case, the partitioned cache does not modify the number of cold misses compared to a general-purpose cache organization.

In comparing our results to related approaches for power optimizations and performance improvement, one can immediately see that the proposed scheme improves both power and performance instead of performing a tradeoff between them. In [11], energy reductions in the range of 10%–25% for 32-kB four-way set-associative caches are reported, yet at the cost of a performance degradation in the range of 2%–6%. In [8], the authors achieve a miss rate reduction in the range of 5%–8%, while introducing additional opcode bits and hardware support for temporal and spatial reuse utilization. Evidently, judicious utilization of application-specific information enables avoidance of this tradeoff space and delivers improvements in both performance and power simultaneously.

VII. CONCLUSION

We have presented a novel methodology for application-specific customization of the cache subsystem of embedded processors in this paper. A precise static analysis of the application has been demonstrated to be capable of identifying the optimal solution for grouping memory access instructions and mapping them to cache partitions with optimal size. Preventing cache interference and cache pollution by utilizing precise application information and subsequently eliminating a large number of power devouring tag operations in addition to reducing the miss rate have constituted the main objectives of the proposed methodology. The achievement of these goals has been confirmed through extensive experimental results. A significant increase in the cache hit rate and a decrease in power consumption have been demonstrated through a representative set of simulation results. The proposed technique has significant implications in system-on-chip designs utilizing embedded-processor cores, as it significantly reduces the number of system bus transactions, thus, resulting in higher system performance and reduced power.

Customizing the embedded-processor architecture utilizing a reprogrammable hardware promises to be a powerful technique toward lower power consumption and higher and deterministic performance in hardware–software systems. At the same time, it helps retain the processor-centric paradigm and extends its advantages to a large class of modern embedded applications.

REFERENCES

[1] W. H. Wolf, "Hardware-software co-design of embedded systems," *Proc. IEEE*, vol. 82, pp. 967–989, July 1992.
[2] J. Henkel and R. Ernst, "A Hardware/Software partitioner using a dynamically determined granularity," in *Proc. 34th Design Automation Conf.*, June 1997, pp. 691–696.
[3] J. *et al.*, "A 160-MHz, 32-b, 0.5-W CMOS RISC microprocessor," in *Proc. IEEE Int. Solid-State Circuits Conf.*, Feb. 1996, pp. 214–229.

[4] M. S. Lam, E. E. Rothberg, and M. E. Wolf, "The cache performance and optimizations of blocked algorithms," in *Proc. 4th Int. Conf. Architectural Support for Programming Languages and Operating Systems*, Apr. 1991, pp. 63–74.

[5] M. E. Wolf and M. S. Lam, "A data locality optimizing algorithm," in *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, June 1991, pp. 30–44.

[6] M. J. Wolfe, "More iteration space tiling," in *Proc. Supercomputing*, Nov. 1989, pp. 655–664.

[7] A. Gonzalez, C. Aliagas, and M. Valero, "A data cache with multiple caching strategies tuned to different types of locality," in *Proc. Int. Conf. Supercomputing*, July 1995, pp. 338–347.

[8] J. Sanchez, A. Gonzalez, and M. Valero, "Static locality analysis for cache management," in *Proc. Conf. Parallel Architectures and Compilation Technique*, Nov. 1997, pp. 261–271.

[9] P. R. Panda and N. D. Dutt, "Low-power memory mapping through reducing address bus activity," *IEEE Trans. VLSI Syst.*, vol. 7, pp. 309–320, Sept. 1999.

[10] N. Bellas, I. Hajj, and C. Polychronopoulos, "Using dynamic cache management techniques to reduce energy in a high-performance processor," in *Proc. Int. Symp. Low Power Electronics and Design*, Aug. 1999, pp. 64–69.

[11] D. H. Albonesi, "Selective cache ways: On-demand cache resource allocation," in *Proc. 32nd Annu. Int. Symp. Microarchitecture*, Nov. 1999, pp. 248–259.

[12] K. Ghose and M. B. Kamble, "Reducing power in superscalar processor caches using subbanking, multiple line buffers and bit-line segmentation," in *Proc. Int. Symp. Low Power Electronics and Design*, Aug. 1999, pp. 70–75.

[13] J. Zhu, P. Agrawal, and D. D. Gajski, "RT level power analysis," in *Proc. Asia South Pacific Design Automation Conf.*, Jan. 1997, pp. 517–522.

[14] N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. S. Kim, and W. Ye, "Energy-driven integrated hardware-software optimizations using simplePower," in *Proc. 27th Annu. Int. Symp. Computer Architecture*, June 2000, pp. 95–106.

[15] D. Burger and T. M. Austin, "The SimpleScalar tool set, version 2.0," Comput. Sci. Dept., Univ. Wisconsin-Madison, Madison, WI, Tech. Rep. 1342, June 1997.

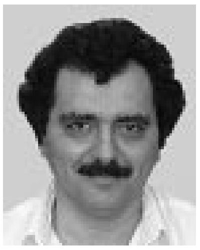
[16] G. Reinman and N. Jouppi, "An integrated cache timing and power model," Western Res. Lab., Palo Alto, CA, Tech. Rep., 1999.

[17] W.-T. Shiue and C. Chakrabarti, "Memory exploration for low power, embedded systems," in *Proc. 36th Design Automation Conf.*, June 1999, pp. 140–145.

[18] S. Nakamura, *Applied Numerical Methods with Software*. Englewood Cliffs, NJ: Prentice-Hall, 1991.

Peter Petrov received the B.S. and M.S. degrees in computer science from Sofia University, Sofia, Bulgaria, in 1996 and 1998, respectively. He is currently working toward the Ph.D. degree in computer engineering at the University of California, San Diego.

His current research interests include application-specific embedded processors, embedded systems, and hardware–software codesign.



Alex Orailoglu (M'84) received the S.B. degree (*cum laude*) in applied mathematics from Harvard University, Cambridge, MA, and the M.S. and Ph.D. degrees in computer science from the University of Illinois, Urbana-Champaign.

From 1983 to 1987, he was a Senior Member of Technical Staff with the Gould Research Laboratories, Rolling Meadows, IL. In 1987, he joined the University of California, San Diego, where he is currently a Professor with the Computer Science and Engineering Department. His current research interests include digital and analog test, fault-tolerant computing, computer-aided design, and embedded processors.

Prof. Orailoglu is a Member of the IEEE Test Technology Technical Council (TTTC) Executive Committee and currently serves as Technical Activities Committee Chair and Planning Co-Chair of TTTC. He serves in numerous technical and organizing committees, including the International Test Conference and the VLSI Test Symposium, and has served as the Technical Program Chair of the 1998 High Level Design Validation and Test (HLDVT) Workshop and as the General Chair of the 1999 HLDVT Workshop.